Help File on C++ and Programming at CSUSM by Rika Yoshii

Compiler versus Editor

empress.csusm.edu is a machine that you remotely log into from anywhere using **putty**,

empress.csusm.edu uses the operating system called Linux which is a type of **UNIX** operating systems.

It is being used by many students at csusm. If you type **users**, you can see who are using the machine at the same time as you are.

FileZilla is a program you can use to upload and download files between your PC and empress.

On empress, you will be using:

g++ - which is a **compiler** that translates C++ into machine code that empress can execute.

C++ can be understood by us but not by empress. Empress needs g++ to translate it into **a.out**.

a.out is specifically produced for empress. It cannot be understood by your PC or Mac which has a different architecture. A compiler is machine dependent.

g++ can check grammar (syntax) errors but cannot check to see if your program makes sense.

Use g++ by listing all the .cpp files you want to compile together. **Never list .h files!**

e.g. g++ hw1.cpp scores.cpp

emacs - which is an **editor** you use to type up programs. Emacs knows what programming language you are using because of the extension part of your file name. e.g. **.cpp**

Emacs can match curly brackets for you and even indent lines perfectly (use Tab on each line).

Emacs cannot compile your program.

Check out other help file links to:

- Learn how to use emacs.
- Learn how to use the UNIX operating system.
- Learn how to use FileZilla.
- Learn how to use G++.

CS311 How to test and debug your program

Testing:

Testing should not be done randomly.

After you finish writing your program, write down **all possible types of cases** that can occur. They correspond to if-then-elses in your program.
For each possible type, write down the actual input.
A very simple example: Compute the average score given the total of scores and the number of students. Case 1: total > 0 and num > 0. Case 2: total = 0 and num > 0 Case 3: total = 0 and num = 0 Case 4: wrong input type for total Case 5: wrong input type for num
A simple example: Add a node to the Ith position of a linked list. Case 1: when the list is empty Case 2: when I = 1, Case 3: when I = 2, Case 4: when I = Count+1, Case 5: when I < 1, Case 6: when I > Count+1

You should not release the software unless it has been tested thoroughly.
Companies hire people to test programs. This is a very important skill.

Debugging:

Do not change code blindly.

While going through the above test cases, if you find a run time error, you need to
1. determine the exact location where the error is occurring
2. find out the values of all relevant variables

To determine the exact location of the error, you have to use temporary couts.
You can mark them so that they are easy to remove later. E.g. cout << "** …" << endl;
Don't forget endl so that the output will appear instantly.

If you don't know which function is causing the errors, please a cout in each function.
After you find out the problem function, place couts within that function.

Then display all relevant variable values to see which one is related to the error.
For the problem variable, you have to find out when and why it took on a wrong value.
i.e. when was the last time the value was changed?

If the problem is due to a pointer (segmentation fault), usually the error line will tell you which pointer is involved. Again, find out when and why it took on a wrong address. i.e. when was the last time the pointer was set or changed?

Compilation Errors

Usually fixing the first compiler error fixes many others. Just go to the line (emacs editor displays line numbers) and fix the syntax (grammar) error. Read carefully, looking for every missing punctuation mark and missing parentheses. At this point, you have to be able to tell wrong C++ syntax from a correct one. If not, study C++ during the summer.

Below you will find some tutorial on basic programming concepts that many students have struggled with. Read the tutorial and use my Visualizers to understand the concepts better.

What are Arrays? How do you use for loops?

When you store data in some organized way for a program to use, it is called a **data structure**.

There are many data structures in C++. One of them is an **array**.

An array is like an upside down building.

Slot 0
Slot 1
Slot 2
Slot 3
Slot 4

Each slot is like a variable and you can place a data item in there.

The type of the data items must be the same for the entire array: all integers, all floats, all strings, etc.

You must name the array first before you can use it and also specify its size.

```
int scores[10];  // declare an array called scores which has 10 slots.  Data items will be integers.

                 // slots are 0 through 9
```

You can access each slot of the array like a variable.

```
scores[0] = 24;

scores[1] = 34;

scores[2] = 32;

sum = scores[1] + scores[2] + scores[0];

cin >> scores[3];

cout << scores[3];
```

Exercise:

Think of kinds of things you may want to store into an array.
Use the simple array Visualizers now.

Using a for-loop with an array:

A big array is difficult to use unless you use a for-loop along with it.

For example, you want to go through every slot from 0 to 99 and do something to each slot.

You can use the for-loop variable to refer to the slot numbers.

```
for (int i = 0; i <= 99; i++)  // i varies from 0 to 99

{     cin << scores[i];  // enter data into the ith slot

}

for (int k = 0; k < 100; k++) // k varies from 0 to 99

{     cout << scores[k] ;  // display what's in slot k

}

for(int j = 0; j < 100; j++)

{ sum = sum + scores[j]; // add jth score to sum }

for (int j = 0; l <=99; j++)

{  if (scores[j] == 100) cout << "Good job student" << j << endl; }
```

Once the scores are in the array, you can search through the array, sort the array, etc.

Later you will learn a variety of sorting algorithms.

Exercise:

Declare an array called prices with 78 slots.

Write a for-loop to go through the 78 slots to find prices > 100 and cout "too expensive- item" followed by the item/slot number.

Use the array with loops Visualizer now.

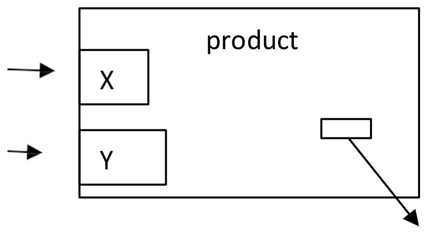
What are Functions?

Let’s say I created a special machine that multiples any two numbers.

I will name this machine **product.**

The product machine has two openings: one labeled X and another labeled Y.

If I place numbers into X and Y, the product will be coming out of the result opening.



Many different people can use this machine for many different purposes.

Mary has Base and Height.

Base [10] $\longrightarrow$ 10 goes into the X slot

Height [20] $\longrightarrow$ 20 goes into the Y slot

John has the Price and Tax-rate.

Price [6.25] $\longrightarrow$ 6.25 goes into the X slot

Taxrate[0.08] $\longrightarrow$ 0.08 goes into the Y slot

Kathy has the Unit Price and the number of items.

UnitPrice [10.3] $\longrightarrow$ 10.3 goes into the X slot

Num [25] $\longrightarrow$ 25 goes into the Y slot

Let’s put this in the context of programming.

The machine product is a **function**. It is created by some programmer.

You can create functions. Some functions have already been created by others.

The input slots X and Y are called **formal parameters**.

Formal parameters have generic names because you don't know what kind of numbers will be coming into the machine/function.

The result will be **returned** from the machine/function.

Mary will **call product**. Mary's Base and Height contents will go into the X and Y slots.

That is called passing the parameters.

Base and Height are the **actual parameters** with very specific names.

When the result is returned, Mary can do anything with it: store it, display it, use it in some other formula.

How complex can a function be? As complex as you want it to be. But the rule of thumb is to not make it too long and make it as generic as possible so that it can be used in many different situations.

Program Examples in C++:

```
// defining the function

float product (float X, float Y)    // float is to say that a float number will be returned

{

   return X * Y ;

}

void Mary ()  // Mary is a client.  Mary calls product.

{ int Base, Height, Area;

   cout << "Enter the base of a rectangle:";

    cin >> Base;

     cout << "Enter the height of the rectangle:";

    cin >> Height;

     Area = product(Base, Height); // think of the red part as being replaced by the value returned

     cout << "The area is: " << Area << endl;

}
```

```
void John ()  // John is a client.  John calls product.
{ float price, taxrate;
  cout << "Enter the price of an item:";
   cin >> price;
    cout << "Enter the tax rate:";
   cin >> taxrate;
    // think of the red part as being replaced by the value returned
    cout << "The cost is: " << product(price, taxrate) << endl;
}
```

Exercises:

1. Draw a machine that divides a number by another number.

2. Draw 2 examples of clients of the machine: what will they enter into the input slots?

    -- what are the formal parameters?

    -- what are the actual parameters?

3. Write the machine as a function in C++

4. Write the clients in C++

A Special Way to Call a Function:

Previously, we talked about calling a function and passing parameters.

We **passed parameters by value** in the above examples.

The contents of Base and Height were placed into the X and Y slots.

What if the contents are too heavy to place into X and Y slots?

What if you wanted more than one value to be returned to you?

That's when you **pass parameters by reference.**

The machine will not have any input slots. The machine will expect you to bring your items in your own boxes. You can even bring an empty box. The machine will work with your boxes.

Product and Sum (bring your boxes)	
Put box1 here	Put box2 here
Put resultbox1 here	Put resultbox2 here

Mary has Base and Height.

Base [10] $\longleftrightarrow$ the box itself goes to product

Height [20] $\longleftrightarrow$ the box itself goes to product

Area [ ] $\longleftrightarrow$ this empty box will be filled with a result

Sum [ ] $\longleftrightarrow$ this empty box will be filled with a result

Program Examples in C++:

```
// defining the function

void product (float& X, float& Y, float& R1, float& R2)    // & indicates pass by reference

{

   R1 = X * Y; // empty box is filled

   R2 = X + Y;   // empty box is filled

}

void Mary ()  // Mary is a client.  Mary calls product.

{ int Base, Height, Area, Sum;

  cout << "Enter the base of a rectangle:";

   cin >> Base;

    cout << "Enter the height of the rectangle:";

   cin >> Height;
  product(Base, Height, Area, Sum); // nothing is returned but Area and Sum are filled

    cout << "The area is: " << Area << endl;

    cout << "The sum is: " << Sum <<< endl;

}
```

Can we mix pass by value and pass by reference? Yes. You can decide per parameter.

If you want to copy the content into an input slot, use pass by value.

If you want to bring your own box and take it back, use pass by reference.

You will learn later what kinds of things are too heavy to be passed by value.

Exercise:

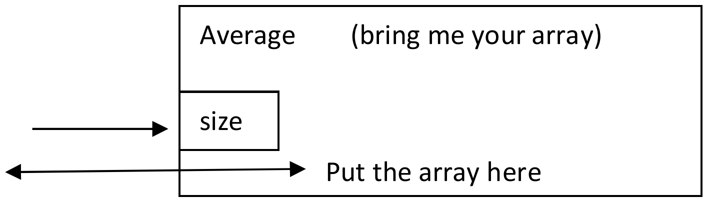
How do you think C++ figures out which actual parameters go with which formal parameters?

Use the pass by value and pass by reference Visualizers now.

Combining Arrays and Functions

Now that you have learned arrays and functions, let's use them together.

The function I want is **average.** Given an array and its size, it will sum up the array contents and return the average.



It turns out that **arrays are always passed by reference automatically in C++. No need to say &.**

```
int average(int nums[], int size)

{  int sum = 0;  // local variable sum

    for (int k = 0; k < size; k++)   // k is also local, k varies from 0 to size-1

   { sum = sum + nums[k]; }

 return sum/size;  }
```

sum and k are variables used by average in order for it to accomplish its task. Nobody else knows about these local variables.

void Mary() // Mary is a client. Mary will call average.

```
{  int scores[78];
   fill(scores, 78); // assume that function fill stored 78 scores into the scores
   array. cout << “The average score was: “ << average(scores, 78) << endl;
 }
```

Exercises:

1. Write John that will fill prices array with 100 prices and then figures out the average price.
2. Draw the function fill. It should ask the user to provide numbers to store into an array. Then write this function in C++.

There are many more Visualizers. As you learn new concepts, please use a Visualizer to understand it better.

Recursion

Recursive functions are one of the easiest concepts to comprehend but one of the hardest to use because people often cannot see a problem in a recursive way.

Mathematical Recursion

Let's first think of recursive math functions since they are the easiest to comprehend.

Factorial of 5 is 5*4*3*2*1.
    Factorial of 4 is 4*3*2*1.
Factorial of n is n*(n-1)*(n-2)*….. 1
    Factorial of n-1 is (n-1)*(n-2)*….. 1

Thus, you can express factorial(n) in terms of factorial(n-1).

factorial(n) = n* factorial(n-1).

But the smallest the n can be is 1. This is the base case.
factorial(1) = 1.

Let's see another math example.
Sum to 5 is 5+4+3+2+1.
    Sum to 4 is 4+3+2+1.
Sum to n is n+(n-1)+(n-2)+….. 1
    Sum to n-1 is (n-1)+(n-2)+….. 1

Thus, sum(n) = n* sum(n-1).
But the smallest the n can be is 1. This is the base case.
sum(1) = 1.

Let's translate these into C++.

Factorial(x) gives an integer. X is an integer. So, we will write.
```
int factorial(int x) {
```

Let's add the base case first.
```
If (x == 1) return 1;  // gives 1
```
Now, add the recursive step.
```
else
        return x*factorial(x-1);  // gives x*factorial(x-1)
```

Therefore, the sum function will be:

```
Int sum(int x)
{
 If (x == 1) return 1;    // base case
 else return x+sum(x-1);   // recursive step
 }
```

To be able to debug recursive functions, you need to **be able to "trace" its execution**. Let's see an example of that.

```
call factorial(5)
      5*factorial(4)
          4*factorial(3)
             3*factorial(2)
                 2*factorial(1)
                     1
```

Each call to factorial is "replaced" by the result of the next line.

```
      120
      5*24
          4*6
             3*2
                 2*1
                     1
```

Exercise 1: Try drawing these for "call sum(4)".
Use the visualizer for math recursion.

Recursion and Arrays

Recursive functions are useful in processing arrays.   Here are some examples:

```
int addup(int A[], int last, int i)  // last is the index for the last slot
{
   If (i == last) return A[i];
   else return A[i]+addup(A, last, i+1);    // add the current element to the rest
}
```

Notice that you need to provide the current slot number as a parameter because if you declare it inside the function, it is a new i each time the function is called.

To invoke it, call it as

```
sum =  addup(MyArray, 5, 0);  // i starts at 0
```

Exercise 2: Try writing a function to multiply all slots of A, and then call it.

The next example does not return anything.

```
void display(int A[], int last, int i)  // last is the index for the last slot
{
   If (i == last) cout << A[i] << endl;;
   else { cout << A[i] << endl;
          display(A, last, i+1); // display the rest
          }
}
```

Call it as

```
display(MyArray, 5, 0);
```

We can pass two arrays and interleave them.

```
void displayInterleaved(int A[], int B[], int last, int i)  // last is the index for the last slot
{
   If (i == last) cout << A[i] << B[i];
   else { cout << A[i] << B[i];
          displayInterleaved(A, B, last, i+1); // display the rest
          }
}
```

Call it as

```
  displayInterleaved(Names, Ages, 20, 0);
```

This function assumes that the two arrays are of the same size. If they are of different sizes, you need to pass “last” for each array.

```
void displayInterleaved(int A[], int B[], int lastA, int lastB, int i)
```

<u>Modifying Arrays</u>

Remember that arrays are passed by reference. If you make a change to an array in a function, it is reflected back in the caller. There is no need to “return” the array.

Here is an example:

```
void swap(int A[], int end, int i)  // swaps adjacent elements of A
{
  if (i == end) return;   // nothing to swap
  else
    { int t = A[i]; A[i]=A[i+1]; A[i+1] = t;  // swap locations i and i+1
      swap(A, end, i+1);   // process the rest
    }
}

int main()
{
  int Ages[5] = {9, 8, 7, 6, 5};
  swap(Ages, 4, 0);

  for (int i = 0; i < 5; i++)
    cout << Ages[i];
}
```

The result will be 87659.

Recursive Algorithms

Many famous algorithms use recursion.

Binary Search algorithm uses recursion.
Famous sorting algorithms such as Merge Sort and Quick Sort use recursion.

In these algorithms, each recursive step makes the array about ½ of its size.

Thus, the start and end of each portion must be passed as arguments.
e.g. int search(int findthis, int A[], int start, int end)   // search for findthis in A in the section indicated and return the location as int
e.g. void sort(int A[], int start, int end) // sort the indicated section

These recursive algorithms are faster than sequential algorithms.

But how do you come up with recursive algorithms?
The key is to think of a problem as sub-problems instead of as sequences of steps.

e.g. to look for Mary in an area, look for her in the left half of the area and then look for her in the right half of the area.

e.g. to travel from SD to NY, travel from SD to LA and travel from LA to NY

Practice viewing problems in a recursive way.

Exercises:

1. Write a recursive function to go through an array one slot at a time to find 100 and return its location.
2. Write a recursive function to go through an array on slot at a time to find the largest number and return it.

Use the visualizer for array recursion.